苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	软件工程	姓名	朱金涛	学号	2327406014
课程名称	操作系统课程实践					成绩	
指导教师	王红玲	同组实验者	无		实验日期	2025 年 9 月 8 日	

实 验 名 称 进程创建

一、 实验目的

1. 加深对进程概念的理解，进一步认识并发执行的实质

2. 掌握 Linux 操作系统的进程创建和终止操作

3. 掌握在 Linux 系统中创建子进程后并加载新映像的操作。

二、 实验内容

1. 编写一个 C 程序，使用系统调用 fork( )创建一个子进程。要求：①在子进程中分别输出当前进程为子进程的提示、当前进程的 PID 和父进程的 PID、根据用户输入确定当前进程的返回值、退出提示等信息。②在父进程中分别输出：当前进程为父进程的提示、当前进程的 PID 和子进程的 PID、等待子进程退出后获得的返回值、退出提示等信息。

2. 编写 C 程序，使用系统调用 fork()创建一个子进程，子进程调用 exec 族函数执行可执行代码 helloworld。

3. 调试并运行下列代码，回答下述问题。

①一共创建了多少个进程？

② 画出创建的进程树。

③ 给出程序执行的结果。

④ 若删除代码中的“wait(&rtn);”语句，程序的运行结果是否相同？为什么？

```
int main()
{
   int rtn=0;
   int i;
   for(i=0;i<2;i++)
   {
      if(fork()==0)
      printf("Child: the parent pid is: %d, the current pid is:%d\n", getppid(),getpid());
```

```
    else
    {
        printf("Parent: the process pid is: %d\n",getpid());
        wait(&rtn);
    }
  }
  exit(0);
}
```

三、 实验步骤和结果

1. 按照题目要求编写程序，调用 fork()函数创建子进程，主要原理就是：

调用 fork()创建子进程，返回值分三种情况

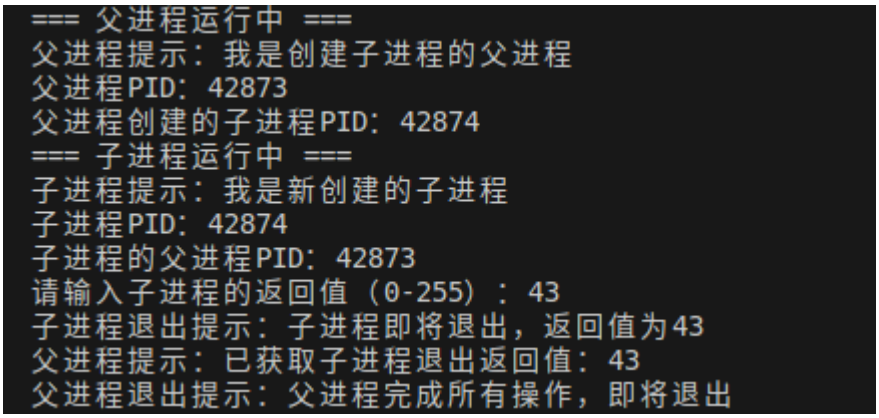
a. 负数：fork 失败

b. 0：当前进程是子进程

c. 正数：当前进程是父进程，返回值为子进程 PID

根据不同返回值可以确定当前处于子进程还是父进程，编写条件判断语句来进行相应处理。

运行结果如下：



注：这里的 43 并不是指上面的 fork()返回值，它只是一个证明父子进程能连通的一个“标志”。

2. 按要求编写两个 c 程序，其中 helloword.c 文件的主函数用于定义输出“Hello, world!”的输出功能，另一个 c 程序中利用 exec 族函数调用 helloword.c 中的程序。

```c
// 1. 调用fork()创建子进程
pid = fork();

// 情况1：fork创建子进程失败
if (pid < 0) {
    perror("fork failed");  // 输出错误原因（如系统资源不足）
    exit(1);                // 父进程异常退出，返回值1表示错误
}

// 情况2：当前进程是子进程（fork返回0）
else if (pid == 0) {
    printf("=== 子进程（PID：%d）运行中 ===\n", getpid());
    printf("子进程：准备调用exec函数执行helloworld...\n");

    // 调用execl()执行helloworld可执行文件
    execl("/home/anders/OS_Operation/op002/ts002/helloworld", "helloworld", NULL);

    // 只有execl()失败时，才会执行下面的错误处理
    perror("execl failed");
    exit(2);
}

// 情况3：当前进程是父进程（fork返回子进程PID）
else {
    printf("=== 父进程（PID：%d）运行中 ===\n", getpid());
    printf("父进程：已创建子进程，子进程PID为：%d\n", pid);
    printf("父进程：等待子进程执行完毕...\n");

    // 等待子进程退出（阻塞父进程）
    wait(&child_status);

    // 解析子进程退出状态，判断exec是否成功
    if (WIFEXITED(child_status)) {
        int exit_code = WEXITSTATUS(child_status);
        if (exit_code == 0) {
            printf("父进程：子进程正常退出，helloworld执行成功！\n");
        } else {
            printf("父进程：子进程异常退出（exec执行失败），退出码：%d\n", exit_code);
        }
    } else {
        printf("父进程：子进程被信号终止（非正常退出）\n");
    }

    printf("父进程：所有操作完成，即将退出\n");
    exit(0);
}
```

运行结果如下：

```
------------------------
Hello World! (来自exec执行的程序)
当前执行进程PID: 44669
------------------------
=== 父进程 (PID: 44668) 运行中 ===
父进程: 已创建子进程，子进程PID为: 44669
父进程: 等待子进程执行完毕...
父进程: 子进程正常退出，helloworld执行成功!
父进程: 所有操作完成，即将退出
```

3. 分析所给代码：首先结合输出结果分析这段代码，结果一共有六行，如下图：

```c
for (i = 0; i < 2; i++)
{
    if (fork() == 0)
    {
        // 子进程：输出父进程PID和自身PID
        printf("Child: the parent pid is: %d, the current pid is: %d\n", getppid(), getpid());
    }
    else
    {
        // 父进程：输出自身PID，并等待当前子进程退出
        printf("Parent: the process pid is: %d\n", getpid());
    }
}
```

wait

```
Parent: the process pid is: 5891
Child: the parent pid is: 5891, the current pid is: 5892
Parent: the process pid is: 5892
Child: the parent pid is: 5892, the current pid is: 5893
Parent: the process pid is: 5891
Child: the parent pid is: 5891, the current pid is: 5894
```

程序中存在循环控制变量 i（如 for i in 0..1），整体流程如下：

首先明确一点：子进程是从 fork()这一行开始执行的。

(1) 父进程调用 fork()创建子进程 1：

1) 由于 fork()给父进程返回子进程 1 的 PID（非 0 值），父进程进入 else 分支，输出第一行结果，随后进入等待状态，等待子进程 1 结束；

2) 由于 fork()给子进程 1 返回 0 值，子进程 1 直接进入 if 分支，输出第二行结果。

(2) 循环迭代至 i=1，子进程 1 继续调用 fork()创建子进程 2：

1) fork()给子进程 1 返回子进程 2 的 PID（非 0 值），子进程 1 进入 else 分支，输出第三行结果，随后进入等待状态，等待子进程 2 结束；

2) fork()给子进程 2 返回 0 值，子进程 2 继承了子进程 1 创建它时的 i=1，进入 if 分支，输出第四行结果后直接结束。

(3) 子进程 2 结束后，子进程 1 的等待完成，随即也结束并回到父进程的等待逻辑中；父进程检测到子进程 1 结束后，循环继续迭代至 i=1，再次调用 fork()创建子进程 3：

1) fork()给父进程返回子进程 3 的 PID（非 0 值），父进程进入 else 分支，输出第五行结果，随后进入等待状态，等待子进程 3 结束；

2) fork()给子进程 3 返回 0 值，子进程 3 继承了父进程创建它时的 i=1，进入 if 分支，输出第六行结果后直接结束。

(4) 子进程 3 结束后，父进程的等待完成，随即也结束，整个程序运行完毕。

现在开始逐一回答问题：

① 一共有四个进程，其中一个父进程，三个子进程。

② 进程树：初始父进程（P0）

```
├── 子进程 1（P1）   // 由 P0 调用 fork() 创建
│    └── 子进程 2（P2）   // 由 P1 调用 fork() 创建
└── 子进程 3（P3）   // 由 P0 调用 fork() 创建
```

③ 运行结果如下：

```
Parent: the process pid is: 5891
Child: the parent pid is: 5891, the current pid is: 5892
Parent: the process pid is: 5892
Child: the parent pid is: 5892, the current pid is: 5893
Parent: the process pid is: 5891
Child: the parent pid is: 5891, the current pid is: 5894
```

④ 删除代码中的 “wait (&rtn);” 语句后，程序的运行结果不相同。因为原程序中 “wait (&rtn);” 的核心作用是让父进程（或子进程 1）阻塞等待对应的子进程（子进程 1 或子进程 2）执行完毕，以此强制实现进程间的执行顺序同步 —— 比如父进程必须等子进程 1 结束才会继续创建子进程 3，子进程 1 必须等子进程 2 结束才会自身结束，最终确保输出顺序固定。而删除 “wait (&rtn);” 后，创建者进程（如父进程、子进程 1）不会再等待子进程，所有进程（初始父进程、子进程 1、子进程 2、子进程 3）会进入无同步的并发执行状态，操作系统会根据进程调度策略（如时间片轮转）随机分配 CPU 执行权，可能出现子进程 2 的输出在子进程 1 之前、子进程 3 的输出在父进程之前等混乱情况，导致最终的输出结果（尤其是输出顺序）与原程序完全不同，且结果具有不确定性。

乱序结果其中之一如下：

```
Parent: the process pid is: 6340
Parent: the process pid is: 6340
Child: the parent pid is: 6340, the current pid is: 6341
Child: the parent pid is: 6340, the current pid is: 6342
Parent: the process pid is: 6341
Child: the parent pid is: 1374, the current pid is: 6343
```

四、 实验总结

本次“进程创建”实验，按要求完成了三项核心内容：一是编写 C 程序通过`fork()`创建子进程，成功在父子进程中分别输出进程标识（PID/PPID）、获取子进程返回值并完成退出处理；二是编写程序结合`fork()`与`exec`族函数，实现子进程加载并执行`helloworld`可执行映像，父进程等待子进程完成并确认执行结果；三是调试特定循环`fork()`代码，明确程序共创建 1 个父进程与 3 个子进程（进程树为初始父进程下含子进程 1 及子进程 1 的子进程 2、父进程的子进程 3），确定了固定输出顺序，且验证了`wait(&rtn)`的进程同步作用——删除该语句后，进程因无阻塞等待进入并发执行状态，受操作系统调度策略影响输出顺序混乱。实验过程中，通过对`fork()`返回值（负数失败、0 为子进程、正数为父进程 PID）的判断、`wait`函数的资源回收与同步功能、`exec`族函数加载新进程映像的实践，不仅达成了加深进程概念理解、掌握 Linux 进程创建/终止及`exec`函数使用的实验目的，更直观认识了进程并发执行的实质与操作系统进程调度对程序运行结果的影响，为后续深入学习操作系统进程管理奠定了实践基础。